Memory management problems

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń
lukasz@coders.school

Łukasz Ziobroń

Not only programming experience:

- C++ and Python developer @ Nokia & Credit Suisse
- Scrum Master @ Nokia & Credit Suisse
- Code Reviewer @ Nokia
- Webmaster (HTML, PHP, CSS) @ StarCraft Area

Training experience:

- C++ trainings @ Coders School
- Practial Aspects Of Software Engineering @ PWr, UWr
- Nokia Academy @ Nokia
- Internal corporate trainings

Public speaking experience:

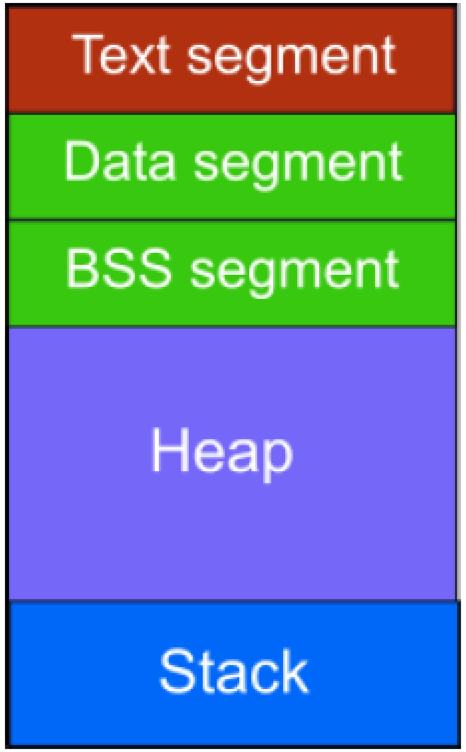
- Academic Championships in Team Programming
- code::dive conference
- code::dive community

Agenda

- Process memory allocation
    - process memory map
    - stack vs heap
    - stack allocation
    - heap allocation
    - new expression and operator new
    - dynamic array allocation
    - dynamic allocation problems
- RAII
- Memory corruption detection

Process memory map

- **text** - the machine instructions
- **data** - initialized static and global data
- **bss** - uninitialized static data
- **heap** - dynamically allocated memory
- **stack** - the call stack, which holds return addresses, local variables, temporary data



Stack vs Heap

Stack

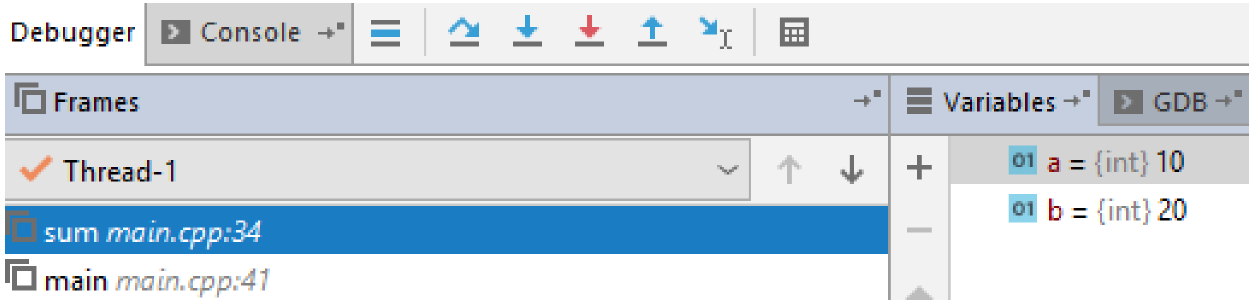
- very fast access
- limit on stack size (OS-dependent)
- not fragmented memory
- automatic memory management (by CPU via Stack Pointer – SP)

Heap

- slower access
- no limit on memory size (OS-managed)
- memory may become fragmented
- manual memory management (allocation and deallocation)

Stack allocation

- A call stack is composed of stack frames
- A stack frame usually includes at least:
  - arguments passed to a function (if any)
  - the return address back to the caller
  - space for local variables (if any)
- Automatic deallocation when out of scope




```cpp
#include <iostream>

int sum(int a, int b)
{
    return a + b;
}

int main()
{
    int a = 10;
    int b = 20;
    std::cout << sum(a, b);
    return 0;
}
```

Stack overflow

- There is a limit on a stack size (OS dependent)

```
int foo()
{
    double x[1048576];
    x[0] = 10;
    return 0;
}

int main()
{
    foo();
    return 0;
}
```

Heap allocation

Heap allocation consists of a few steps

- pointer allocation on a stack
- sizeof(T) bytes allocation on a heap
- T's constructor call on allocated memory
- the memory address assignment to the pointer

- manual deallocation using delete operator

```
void heap()
{
    int *p = new int(100);
    delete p;
}

void heap()
{
    int *p;
    p = new int;
    *p = 100;
    delete p;
}
```

New expression and operator new

new expression does 3 things

- sizeof(T) bytes allocation on a heap (via proper `operator new`)
- T's constructor call on allocated memory
- the memory address assignment to the pointer

```
// replaceable allocation functions
void* operator new  ( std::size_t count );
void* operator new[]( std::size_t count );
// replaceable non-throwing allocation functions
void* operator new  ( std::size_t count, const std::nothrow_t& tag);
void* operator new[]( std::size_t count, const std::nothrow_t& tag);
// user-defined placement allocation functions
void* operator new  ( std::size_t count, user-defined-args... );
void* operator new[]( std::size_t count, user-defined-args... );
// additional param std::align_val_t since C++17, [[nodiscard]] since C++20
// and some more versions on https://en.cppreference.com/w/cpp/memory/new/operator_new
```

Dynamic array allocation

- delete[] is used to free an array memory

```cpp
#include <iostream>

int main() {
    int staticArray[] = {1, 2, 3, 4, 5, 6};

    constexpr auto size = 10;
    int* dynamicArray = new int[size];
    for (int i = 0; i < size; ++i)
        *(dynamicArray + i) = i * 10;

    for (int i = 0; i < size; ++i)
        std::cout << dynamicArray[i] << '\n';

    delete[] dynamicArray;
}
```

Dynamic allocation problems

- accessing out-of-bounds memory
- dangling pointer
- double deleting
- null pointer dereference
- freeing memory blocks that were not dynamically allocated
- freeing a portion of a dynamic block
- memory leak

These problems can be addressed by ASAN (Address Sanitizer) or Valgrind.
Unfortunately they do not work on Windows ☹

Accessing out-of-bounds memory

Undefined behavior

```cpp
#include <iostream>

int main() {
    const auto size = 10;
    int* dynamicArray = new int[size];
    for (int i = 0; i <= size; ++i)
        *(dynamicArray + i) = i * 10;

    for (int i = 0; i <= size; ++i)
        std::cout << dynamicArray[i] << '\n';

    delete[] dynamicArray;
}
```

Dangling pointer

Pointer which indicate to something that is not valid

```cpp
#include <iostream>

struct Msg {
    int value{100};
};

void processMsg(Msg* msg) {
    std::cout << msg->value << '\n';
}

int main() {
    Msg* m = new Msg();
    // ...
    delete m;

    processMsg(m);

    return 0;
}
```

Double delete

Happens when a dangling pointer is deleted

```
class Msg {};

void processMsg(Msg* msg) {
    // ...
    delete msg;
}

int main() {
    Msg* m = new Msg{};
    processMsg(m);
    delete m;
}
```

Null pointer dereference

Happens when a nullptr is used

```cpp
#include <iostream>

int main() {
    int* p = new int{10};
    delete p;
    p = nullptr;

    std::cout << *p << '\n';

    return 0;
}
```

Freeing stack allocated blocks

```
class Msg {};

void processMsg(Msg* msg) {
    // ...
    delete msg;
}

int main() {
    Msg m;
    processMsg(&m);

    return 0;
}
```

Freeing a portion of a dynamic block

Using delete instead of delete[]

```
int main() {
    constexpr auto size = 4u;
    int* array = new int[size]{1, 2, 3, 4};
    delete array;

    return 0;
}
```

Memory leak

Allocated memory which cannot be freed because threre is no pointer that points to it

```cpp
#include <iostream>

int main() {
    int* p = new int{10};
    p = new int{20};
    std::cout << *p << '\n';
    delete p;

    return 0;
}
```

A simple question...

How many possible path of execution are here?

```
String EvaluateSalaryAndReturnName(Employee e)
{
    if( e.Title() == "CEO" || e.Salary() > 100000 )
    {
        cout << e.First() << " " << e.Last()
             << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
```

- 23 (twenty three)
- Exceptions are the reason
- Example by Herb Sutter, GotW#20

RAII

- Resource Acquisition Is Initialization
  - idiom / pattern in C++
  - each resource has a handler
  - acquired in constructor
  - released in destructor

- Benefits
  - shorter code (automation)
  - clear responsibility
  - applies to any resources
  - no need for finally sections
  - predictable release times
  - language-level guarantee of correctenss

	acquire	release
memory	new, new[]	delete, delete[]
files	fopen	fclose
locks	lock, try_lock	unlock
sockets	socket	close

Memory corruption detection

- Address Sanitizer (ASAN)
  - add a compilation flags –fsanitize=address -g
  - run a binary
- Valgrind
  - compile a binary
  - run a binary under valgrind:
    - valgrind /path/to/binary
  - use additional checks:
    - valgrind --leak-check=full /path/to/binary

Both does not work on Windows ☹

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń
lukasz@coders.school